Compositional Verification of Self-Adaptive Cyber-Physical Systems

Conference or Workshop Item

Compositional Verification of Self-Adaptive Cyber-Physical Systems*

Aimee Borda
Trinity College Dublin
Dublin, Ireland
bordaa@tcd.ie

Liliana Pasquale
University College Dublin
Dublin, Ireland
Liliana.Pasquale@ucd.ie

Vasileios Koutavas
Trinity College Dublin
Dublin, Ireland
Vasileios.Koutavas@scss.tcd.ie

Bashar Nuseibeh
Open University, UK
Lero, Ireland
b.nuseibeh@open.ac.uk

ABSTRACT

Cyber-Physical Systems (CPSs) must often self-adapt to respond to changes in their operating environment. However, using formal verification techniques to provide assurances that critical requirements are satisfied can be computationally intractable due to the large state space of self-adaptive CPSs. In this paper we propose a novel language, *Adaptive CSP*, to model self-adaptive CPSs modularly and provide a technique to support compositional verification of such systems. Our technique allows system designers to identify (a subset of) the CPS components that can affect satisfaction of given requirements, and define adaptation procedures of these components to preserve the requirements in the face of changes to the system's operating environment. System designers can then use *Adaptive CSP* to represent the system including potential self-adaptation procedures. The requirements can then be verified only against relevant components, independently from the rest of the system, thus enabling computationally tractable verification. Our technique enables the use of existing formal verification technology to check requirement satisfaction. We illustrate this through the use of FDR, a refinement checking tool. To achieve this, we provide an adequate translation from a subset of *Adaptive CSP* to the language of FDR. Our technique allows system designers to identify alternative adaptation procedures, potentially affecting different sets of CPS components, for each requirement, and compare them based on correctness and optimality. We demonstrate the feasibility of our approach using a substantive example of a smart art gallery. Our results show that our technique reduces the computational complexity of verifying self-adaptive CPSs and can support the design of adaptation procedures.

*This work was supported by Science Foundation Ireland grants 13/RC/2094 (Lero) and 15/SIRG/3501 and European Research Council Advanced Grant no. 291652 (ASAP).

SEAMS '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5715-9/18/05…$15.00
https://doi.org/10.1145/3194133.3194146

CCS CONCEPTS

• **Software and its engineering → System modeling languages**; **Formal software verification**; • **Theory of computation →** *Process calculi*;

ACM Reference Format:
Aimee Borda, Liliana Pasquale, Vasileios Koutavas, and Bashar Nuseibeh. 2018. Compositional Verification of Self-Adaptive Cyber-Physical Systems. In *SEAMS '18: SEAMS '18: 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3194133.3194146

1 INTRODUCTION

Computational and communication capabilities are being embedded into physical entities and processes, resulting in a proliferation of Cyber-Physical Systems (CPSs), such as smart buildings, autonomous vehicles and smart cities. CPSs must be *resilient*—that is, they must satisfy their requirements in the face of changes in their operating environment. To achieve this aim, CPSs may self-adapt when their environment changes, for example by changing their behaviour [7]. Self-adaptive CPSs increasingly support critical services, requiring assurances to be provided to guarantee that key requirements are indeed satisfied in the presence of self-adaptation. However, the highly dynamic nature of self-adaptive CPSs can make assurances hard to establish. Moreover, use of formal verification techniques to ensure satisfaction of critical requirements can be computationally intractable due to the large state space of self-adaptive CPSs.

Techniques for modelling self-adaptive systems using rigid modules (e.g., [21]) do not work well for CPSs where different groupings of concurrent cyber and physical components may need to be considered to satisfy each system requirement. Existing work to verify self-adaptive CPS [36] uses explicit state model checking. This can lead to state explosion and may not be computationally feasible for large-scale systems.

In this paper we tackle the complexity of self-adaptive CPSs by proposing a novel language, *Adaptive CSP* (*ACSP*), to represent modularly and support *compositional verification* of self-adaptive CPSs. We also provide a technique to do this and explore alternative adaptation procedures that are modelled using our language.

First, a system designer should model the components of a CPS as parallel processes in *ACSP*. Then she should identify a sub-set of

these components that can affect satisfaction of each given requirement, and encode an adaptation of these components to preserve the requirement. Although manual, the task of identifying components relevant to a requirement can be guided by topological relationships [31] (e.g., containment and connectivity between system components). For example, in a smart building a valuable physical asset may be contained in a room while a digital asset, such as data, could be contained in a server. Moreover, physical areas can be connected through doors or digital devices can be connected through an IP network. If a requirement aims to preserve integrity of digital information stored in a server, the components that can affect its satisfaction are, for example, the rooms that are connected to the one containing the server as well as the devices digitally connected to the server.

Similarly to existing standard process languages (e.g., [25, 30]) *ACSP* can encode system components as parallel processes and system actions (e.g., access/exit to/from a room, connection to the wireless network) as *first-order* communication (i.e., transmission of data). *ACSP* also adds significant expressive power compared to standard process languages by encoding adaptation functionality through a new, higher-order communication construct. More precisely, it allows representing system monitors which use *higher-order* communication (i.e., transmission of processes) to trigger adaptation of components.

System actions as well as adaptation functionalities can be verified for the modelled components independently from the entire CPS thus enabling compositional verification. Because the state space of such sets of components can be significantly smaller than that of the entire system, it is possible to use formal verification technologies. In this paper we use FDR [20], a refinement checking tool for CSP; however, our technique is general enough to allow the use of other verification technologies for process calculi, such as (bi-)simulation, testing preorder, and modal logic techniques (e.g., [4, 12, 15, 24, 30, 32]). If the verification fails, the system designer can explore alternative adaptation procedures, which can be implemented at a different granularity, i.e. across a fewer or more CPS components. We showcase and evaluate our approach using a running example of a smart art gallery. Our results show that our technique reduces significantly the computational complexity of verifying self-adaptive CPSs. Moreover, the ability to explore different adaptation procedures allows identifying and selecting one that maintains satisfaction of critical requirements.

The rest of the paper is organised as follows. The next section discusses an art gallery building as a motivating example of a self-adaptive CPS. Section 3 describes our technique for modelling and verifying such systems using a novel process language *ACSP*, and standard verification tools such as FDR. Section 4 presents *ACSP* and Section 5 the encoding of the art gallery in this language. We then discuss how *ACSP* processes can be translated and verified in FDR. Sections 7 and 8 discuss related work and conclusions, respectively.

2 THE ART GALLERY EXAMPLE

Our motivating example is set in an art gallery building. The two-floors plan of the gallery is shown in Figure 1. *Floor 1* includes a corridor and a set of exhibition areas (*Hall* and rooms *A*, *B*, *C*, *D*)

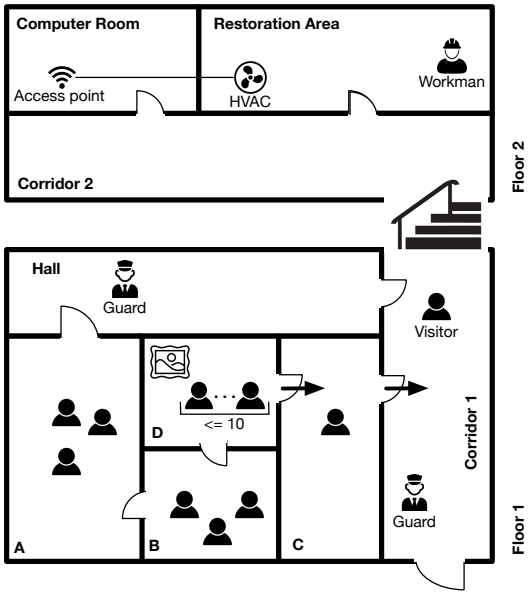

Figure 1: Two-floors plan of the Art Gallery.

where paintings are displayed. *Floor 2* includes a *Restoration Area* where maintenance and restoration of artwork is carried out. An *HVAC* maintains a predefined target temperature and humidity level in the *Restoration Area*. A wireless access point in the *Computer Room* provides internet connectivity to the devices located in *Floor 2*. It is also connected to the *HVAC* allowing the latter to be monitored and controlled remotely.

Public access to the *Restoration Area* is allowed to promote the restoration activities of the gallery. Nevertheless, a critical security requirement for the art gallery is:

Requirement 1. *Visitors should not interfere with the restoration process.*

Req. 1 could be violated when a visitor is in the *Restoration Area* without a guard. However, controlling access to the *Restoration Area* is not always possible. For example, when restoration work is performed, access to the restoration area should be free of controls to allow efficient movement of workers. This can lead to the situation where visitors can access the *Restoration Area* and interfere with the restoration work. To avoid violating Req. 1 a system designer can introduce adaptation procedures at different granularities. For example, adaptation can be applied at the level of the second floor, allowing access to the *Restoration Area* only with a guard. However, this would not be optimal because it would obstruct movement of workers. Alternatively, an adaptation at a coarser granularity could also consider *Corridor 1* and the *Stairs*. This can force the guard located in *Corridor 1* to escort visitors as soon as they start climbing the stairs to reach the second floor. This would ensure that visitors always reach the *Restoration Area* accompanied by a guard.

In our example, a very important painting at the core of the current exhibition is located in *Room D*. Visitors can enter this room from *Room B* and exit to *Room C*. To maintain integrity of the painting, the following requirement should be satisfied.

Requirement 2. *No more than 10 visitors should be in* Room D *at the same time.*

To satisfy this requirement an adaptation can be applied only to *Room D*: visitor access to this room can be allowed only when there are less than 10 people in it. However, there are situations in which Req. 2 can be violated, for example when multiple visitors enter *Room D* at the same time (e.g., by tailgating). Including *Rooms A* and *B* in the design of the adaptation procedure can support satisfaction of Req. 2. It is possible to ensure that the total number of visitors in *Rooms A*, *B*, and *D* does not exceed the maximum number of people allowed in *Room D* alone, and require a guard in the *Hall* to prevent tailgating when visitors enter *Room A*. Alternatively, an adaptation procedure could keep track of people in *Rooms A*, *B* and *D* and allow movement from one room to the next only when the sum of people in both rooms is less than an upper bound. For example, movement from *Room B* to *D* would be allowed only when the people in the two rooms are less than 10.

Finally, the following requirement should be satisfied.

Requirement 3. *The HVAC should not be controlled remotely by unauthorised users.*

Considering the vulnerabilities of the wireless protocols, Req. 3 can be violated when a malicious visitor's device connects to the wireless network and takes control of the *HVAC*. To satisfy this requirement an adaptation procedure can be designed by considering the *Restoration Area*, the *Computer Room* and *Corridor 2*. In this case, an adaptation procedure can disconnect the *HVAC* from the network when an untrusted device in the second floor is connected to the *Access Point*. Additionally, this adaptation procedure could disconnect clients and reconnect the *HVAC* to the *Access Point* when a designated trusted device connects, presumably to perform maintenance operations on the *HVAC*.

This example demonstrates how adaptation procedures aiming at preserving a critical requirement can be implemented at different granularities. Note that dividing the system in rigid self-adaptive modules *a priori* poses the risk of not being able to implement an optimal adaptation procedure. Therefore, we provide a methodology to support system designers in the definition of adaptation procedures at appropriate granularities, which satisfy specific requirements. The benefit of discovering a good level of granularity for an adaptation procedure is that it can be verified only in relation to the components it affects, ignoring the rest of the system. This makes it more tractable to use formal techniques to provide assurances for the system, as we later show in this paper.

3 MODELLING AND VERIFYING SELF-ADAPTIVE CPSs

Our technique to model self-adaptive CPSs and verify their requirements comprises the following steps.

Step 1: modelling the CPS. A system designer first identifies the main cyber and physical components of the CPS, such as rooms and assets. Each component may have containment and connectivity relationships with other components. Fig. 2 represents the components of the art gallery example and their relationships. For example, *Corridor 2* can contain agents (e.g., visitors, employees and guards) and it is physically connected to the *Stairs*, the *Restoration Area*, and the *Computer Room*. The *Access Point* can have connectivity relationships with the *HVAC* and with other employees' and visitors' devices located in *Floor 2* that are connected to the wireless network. The *Computer Room* has a containment relationship with the *Access Point* and the agents that are located in it. Similarly, the *Restoration Area* has containment relationship with the *HVAC* and the agents in it.

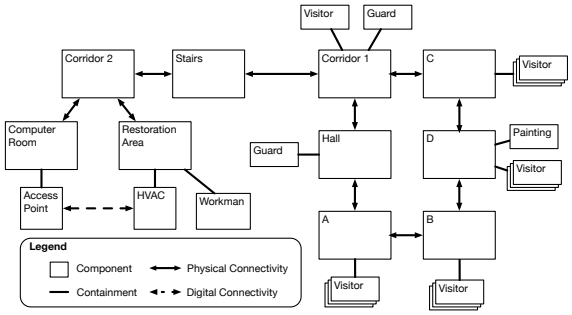

Figure 2: Components of the Art Gallery.

Topological relationships can enable execution of system actions. For example, if an agent is contained in a physical area, she can access other connected physical areas and perform actions in them. In the art gallery example, if an agent moves from the *Stairs* to *Corridor 2*, she can connect a device she is carrying to the *Access Point*. Moreover, agents contained in a physical areas can access and/or control co-located assets, if available. For example, agents in the *Computer Room* can switch on/off the *Access Point*.

Step 2: exploring adaptation procedures. In this step, the system designer examines each requirement, determining the components that can affect its satisfaction. Here we consider this to be a manual task, but it can be guided by the topology of the CPS. Certainly, relevant components include the one the requirement refers to. They may also include additional components identified considering its containment and connectivity relationships. For example, the components affecting satisfaction of Req. 1 can include the *Restoration Area* and those related to it (*Corridor 2*, *Stairs* and *Corridor 1*). Different adaptation procedures can be explored starting from the most specific component relevant to the requirement, and including more components affecting the requirement's satisfaction as necessary. For example, to satisfy Req. 1, as we discussed in the previous section, an adaptation procedure may consider the *Restoration Area* and *Corridor 2*. More precisely, it may keep track of the presence of a guard in *Corridor 2*, and only allow access/exit to/from the *Restoration Area* if a guard is in *Corridor 2*. However, this may not always be desirable because when a guard is not in *Corridor 2*, workers may not be allowed to freely access or leave the *Restoration Area*. Alternatively, the adaptation procedure could monitor visitor access to the *Stairs* from *Corridor 1* and notify the guard stationed in *Corridor 1* to escort visitors in *Corridor 2*.

Step 3: encoding with Adaptive CSP. The system designer can subsequently encode the CPS components and adaptation procedures in our *ACSP* language. Each component can be encoded as a parallel process or an internal state. In our example, we encode all components except agents as processes. Each component representing

a physical location has an internal state representing the number of visitors, guards and workers that are contained in it. System actions, such as agent movements or *Access Point* connections, are implemented as first-order communication events (i.e. transmission of data). Adaptation procedures are composed of a monitor, which listens to first-order communication events and issues higher-order communication events, encoding the the adaptation procedure selected in the previous step. As discussed in the following sections, to tag the components that can be adapted we place them inside *named locations*.

Step 4: verifying system requirements. Finally, the system designer verifies each requirement. This is done by translating *parts* of the self-adaptive CPS, as encoded in our language, to existing verification tools—here we choose FDR. The smallest set of components that need to be translated is easy to find. For each requirement, only the following elements are translated to FDR:

(1) The components that affect the satisfaction of the given requirement (i.e., those explicitly named by the requirement). If these components are inside named *ACSP* locations (i.e., they are adaptable), then the entire location must be translated, potentially including more components.
(2) All the adaptation procedures that affect the locations included in the translation. Here we assume that each adaptation procedure affects exactly one location, but many procedures can affect the same location.
(3) The first-order communication events of the components included in the translation.

In this manner we translate the smaller set of components that include those relevant to the requirement, and whose higher-order adaptation events can be entirely internalised (no adaptation event can cross the boundaries of this set) and encoded as internal transitions in FDR. The requirement itself is encoded using the capabilities of the verification tool; in FDR we use refinement. This is described in more detail in Section 6. If the verification fails then the requirement and adaptation procedure must be re-examined, going back to *Step 2*, possibly considering a different granularity for the adaptation procedure.

4 THE PROCESS LANGUAGE *ACSP*

Our technique is based on a process language which we call *Adaptive CSP* (*ACSP*). This language extends Communicating Sequential Processes (CSP) [25] with locations and higher-order communication. *ACSP* is defined over a set of *first-order events*, ranged over by e, and a set of *location names*, ranged over by l. We reserve the special first-order event τ for internal communication, and use c, d to range over events and location names.

$$c, d ::= e \mid l$$

We also use X for process variables and y for event variables; we let $\hat{e}$ range over events and event variables. We use standard notation for sequences ($\vec{\cdot}$) and the usual syntactic sugar from process languages.

The abstract syntax of the language is:

$$P,Q,R,M,N,\Pi ::= SKIP \mid \Box_{i\in\mathcal{I}}\, \hat{e}_i{\rightarrow}P_i \mid \text{if } \hat{e}_1 \le \hat{e}_2 \text{ then } P \text{ else } Q$$
$$\mid P \underset{\Sigma}{\parallel} Q \mid (\nu c)P \mid X(\vec{\hat{e}}) \mid \mathrm{rec}X(\vec{y} := \vec{\hat{e}}).P \mid l!P.Q \mid l\langle P\rangle$$

Processes, ranged over by P, Q, R, M, N, Π include the standard CSP processes: the inactive process (*SKIP*); *external choice* where the environment can choose between a set of events $\{e_i \mid i \in \mathcal{I}\}$, written as $\Box_{i\in\mathcal{I}}\, e_i{\rightarrow}P_i$, with $\mathcal{I}$ being an indexing set; a conditional process (if $e \le e'$ then P else Q); parallel composition of processes ($P \underset{\Sigma}{\parallel} Q$) which synchronise on a set of non-τ events Σ; scope restriction of events and location names ($(\nu c)P$); and recursion ($\mathrm{rec}X(\vec{y} := \vec{e}).P$) through process variables (X) and event parameter variables ($\vec{y}$). *ACSP* processes also include the new constructs of a higher-order prefix $l!P.Q$ that sends a process P to location l, and named locations ($l\langle P\rangle$). Intuitively, a named location $l\langle P\rangle$ marks process P with name l, which can be adapted to $l\langle Q\rangle$ through a higher-order communication with a prefix $l!Q.R$, residing outside of the location.

The operational semantics of *ACSP* is defined by labelled transitions annotated by: τ, when the transition is internal; $e \neq \tau$, when it is a first-order synchronisation event; $l?P$, when location l is being adapted and becomes P; $l!P$, when an external process initiates such an adaptation. Higher-order transition annotations are ranged over by h; all transition annotations are ranged over by α.

$$h ::= l!P \mid l?P \qquad\qquad \alpha ::= e \mid h$$

The rules of the transition semantics of *ACSP* are shown in Fig. 3. Adaptation transitions are the main novelty of *ACSP*; event transitions are similar to CSP. External choice (EvCh) can transition to one of its residual processes P_j, annotating the transition with the corresponding action e_j. The parallel rule (EvParL) and its (omitted) symmetric rule propagate an event transition e of P over a parallel composition $P \underset{E}{\parallel} Q$, provided that e is not mentioned in the set of events E on which P and Q must synchronise. Rule EvSync synchronises such an event.

Rule Rec unfolds a recursion by an internal transition during which the formal parameters of the recursive process $\vec{y}$ are replaced by the actual parameters $\vec{e}$, and the recursion variable X is replaced with the recursive process itself. Note that substitution of X in processes of the form $X(\vec{e'})$ preserves $\vec{e'}$ as the formal parameters. That is, $X(\vec{e'})[{}^{\mathrm{rec}X(\vec{y}:=\vec{e}).P}/_{X}]$ becomes $\mathrm{rec}X(\vec{y} = \vec{e'}).P$, and we can have the following example transitions:

$$\mathrm{rec}X(y := 1).(e.y{\rightarrow}X(y+1)) \xrightarrow{\tau}$$
$$e.1{\rightarrow}\mathrm{rec}X(y := 1+1).(e.y{\rightarrow}X(y+1)) \xrightarrow{e.1} \xrightarrow{\tau}$$
$$e.2{\rightarrow}\mathrm{rec}X(y := 2+1).(e.y{\rightarrow}X(y+1)) \xrightarrow{e.2} \xrightarrow{\tau} \ldots$$

Rule EvHide hides an event from the surrounding processes, converting it into τ; EvEsc propagates an event over a scope restriction, and IfTrue and (omitted) IfFalse evaluate a conditional. Our language also includes rule EvLoc which propagates event transitions over locations.

We have chosen CSP-style synchronisation for first-order events in our language because they simplify the encoding of *monitor processes*, which can be used to keep track of state and encode adaptation procedures.

Example 4.1. Consider *Corridor 2* (c_2) from Fig. 1, which is connected with a door to the *Stairs* (s). We can encode the movement of visitors, employees and guards from one space to the other by

Event Transitions:

$$\textsc{EvCh}\ \frac{j \in \mathcal{I}}{\Box_{i\in\mathcal{I}}\, e_i{\rightarrow}P_i \xrightarrow{e_j} P_j} \qquad \textsc{EvParL}\ \frac{P \xrightarrow{e} P' \quad e \notin E}{P \underset{E}{\parallel} Q \xrightarrow{e} P' \underset{E}{\parallel} Q} \qquad \textsc{EvSync}\ \frac{P \xrightarrow{e} P' \quad Q \xrightarrow{e} Q' \quad e \in E \quad e \neq \tau}{P \underset{E}{\parallel} Q \xrightarrow{e} P' \underset{E}{\parallel} Q'} \qquad \textsc{Rec}\ \frac{\sigma = [\vec{e}/\vec{y}][^{(\mathrm{rec}X(\vec{y}:=\vec{e}).P)}/_X]}{\mathrm{rec}X(\vec{y}:=\vec{e}).P \xrightarrow{\tau} P\sigma}$$

$$\textsc{EvHide}\ \frac{P \xrightarrow{e} P'}{(\nu e)P \xrightarrow{\tau} (\nu e)P'} \qquad \textsc{EvEsc}\ \frac{P \xrightarrow{e} P' \quad c \neq e}{(\nu c)P \xrightarrow{e} (\nu c)P'} \qquad \textsc{IfTrue}\ \frac{e_1 \leq e_2}{(\text{if } e_1 \leq e_2 \text{ then } P \text{ else } Q) \xrightarrow{\tau} P} \qquad \textsc{EvLoc}\ \frac{P \xrightarrow{e} P'}{l\langle P\rangle \xrightarrow{e} l\langle P'\rangle}$$

Adaptation Transitions:

$$\textsc{AdSnd}\ \frac{}{l!P.Q \xrightarrow{l!P} Q} \qquad \textsc{AdRcv}\ \frac{}{l\langle Q\rangle \xrightarrow{l?P} l\langle P\rangle} \qquad \textsc{AdSyncL}\ \frac{P \xrightarrow{l!R} P' \quad Q \xrightarrow{l?R} Q'}{P \underset{E}{\parallel} Q \xrightarrow{\tau} P' \underset{E}{\parallel} Q'} \qquad \textsc{AdParL}\ \frac{P \xrightarrow{h} P'}{P \underset{E}{\parallel} Q \xrightarrow{h} P' \underset{E}{\parallel} Q} \qquad \textsc{AdEsc}\ \frac{P \xrightarrow{h} P' \quad c \notin h}{(\nu c)P \xrightarrow{h} (\nu c)P'} \qquad \textsc{AdLoc}\ \frac{P \xrightarrow{h} P'}{l\langle P\rangle \xrightarrow{h} l\langle P'\rangle}$$

Figure 3: Transition Semantics of *ACSP* (omitting symmetric rules).

a family of first-order events: $vis_{(s,c_2)}$ encodes the movement of a visitor from the *Stairs* to *Corridor 2*, and $vis_{(c_2,s)}$ the reverse; similarly, events $grd_{(s,c_2)}$ and $grd_{(c_2,s)}$ encode the movement of guards between the two spaces. A process that models visitor movement is the following:

$$C2_0 = \Box \begin{cases} vis_{(s,c_2)}{\rightarrow}C2_0 \\ vis_{(c_2,s)}{\rightarrow}C2_0 \end{cases}$$

Here we assume that a guard is already in, and not allowed to leave c_2. In order to keep track of the number of visitors in *Floor 2*, entering from the stairs, we can use a monitor process for the events $vis_{(s,c_2)}$ and $vis_{(c_2,s)}$:

$$C(v) = \Box \begin{cases} vis_{(s,c_2)}{\rightarrow}C(v+1) \\ vis_{(c_2,s)}{\rightarrow}C(v-1) \\ cnt.v{\rightarrow}C(v) \end{cases}$$

Note that here we write $C(v) = P$ instead of $C = \mathrm{rec}X(y := v).P$, and assume a standard encoding of natural numbers. Process $C(v)$ keeps the number of visitors in v, which it can report through the (parameterised) event $cnt.v$. We can now compose the two processes

$$C2_0 \underset{vis}{\parallel} C(v)$$

such that whenever a visitor enters or leaves *Corridor 2* from the *Stairs*, the state of $C(v)$ increments or decrements, accordingly. □

Adaptation is a higher-order transition which has a single sender and a single receiver. The sender is a process $l!P.Q$ which performs a transition annotated with $l!P$, according to rule AdSnd of Fig. 3. The receiver is a location with name l which performs transition $l?P$, according to rule AdRcv. These transitions synchronise with rule AdSyncL (and its omitted symmetric rule), and are propagated over parallel, scope restriction, and locations according to rules AdParL (and its omitted symmetric), AdEsc, and AdLoc, respectively. Because of the sender-receiver communication pattern of adaptation, and since we intend locations to have unique names, we choose binary communication for these higher-order transitions.

Example 4.2. Continuing from Example 4.1, an adaptation procedure can query the counter after every visitor move, and change the behaviour of *Corridor 2*, when for example there are no more visitors in *Floor 2*. To achieve this we consider that process $C2_0$, encoding visitor movement in and out of c_2, is inside a location l_{C2}, and can thus be adapted. The following process Π installs process $C2_1$ in this location when all visitors have left c_2:

$$\Pi = vis_{(\_,\_)}{\rightarrow}cnt.v{\rightarrow}\,\text{if } v = 0 \text{ then } l_{C2}!C2_1.\Pi \text{ else } \Pi$$

Here $vis_{(\_,\_)}$ represents any visitor events. Process $C2_1$ allows the guard to leave c_2 and prevents further movement into c_2, encoding the closing of the space.

$$C2_1 = grd_{(c_2,s)}{\rightarrow}SKIP$$

The model of *Corridor 2* is the composition of the above processes.

$$(\nu\, l_{C2})(\, l_{C2}\langle C2_0\rangle \underset{vis}{\parallel} C(1) \underset{vis,cnt}{\parallel} \Pi\,)$$

The following execution shows how location l_{C2} is adapted when the last visitor leaves:

$$(\nu\, l_{C2})(\, l_{C2}\langle C2_0\rangle \underset{vis}{\parallel} C(1) \underset{vis,cnt}{\parallel} \Pi\,) \xrightarrow{vis_{(c_2,s)}}$$

$$(\nu\, l_{C2})(\, l_{C2}\langle C2_0\rangle \underset{vis}{\parallel} C(0) \underset{vis,cnt}{\parallel} \Pi_1\,) \xrightarrow{cnt.0}$$

$$(\nu\, l_{C2})(\, l_{C2}\langle C2_0\rangle \underset{vis}{\parallel} C(0) \underset{vis,cnt}{\parallel} \Pi_2(0)\,) \xrightarrow{\tau}$$

$$(\nu\, l_{C2})(\, l_{C2}\langle C2_0\rangle \underset{vis}{\parallel} C(0) \underset{vis,cnt}{\parallel} l_{C2}!C2_1.\Pi\,) \xrightarrow{\tau}$$

$$(\nu\, l_{C2})(\, l_{C2}\langle C2_1\rangle \underset{vis}{\parallel} C(0) \underset{vis,cnt}{\parallel} \Pi\,)$$

Here we let $\Pi_1 = cnt.v{\rightarrow}\Pi_2(v)$ and $\Pi_2(v) = (\text{if } v = 0 \text{ then } l_{C2}!C2_1.\Pi \text{ else } \Pi)$. The last transition is due to the synchronisation of the transitions

$$l_{C2}\langle C2_0\rangle \xrightarrow{l?C2_1} l_{C2}\langle C2_1\rangle \qquad\qquad l_{C2}!C2_1.\Pi \xrightarrow{l!C2_1} \Pi$$

Note that in this example we restricted the scope of the l_{C2} to illustrate that the adaptation of a location can be *localised*—no process outside the restriction can adapt l_{C2}. □

In the following we will use functions $out(P)$ and $in(P)$ to return the set of free location names in P used in send-prefixes and locations, respectively.

Definition 4.3. $out(P)$ and $in(P)$ are defined by the rules

$$out((\nu c).P) = out(P) - \{c\}$$
$$out(l!P.Q) = \{l\} \cup out(P) \cup out(Q)$$
$$in(l\langle P\rangle) = \{l\}$$
$$in((\nu c).P) = in(P) - \{c\}$$

and in all other constructs of the language by the union of the results of recursive calls to these functions. □

The language *ACSP* is powerful enough to support nested locations, adaptation procedures within locations (which can themselves be adapted) and location redundancy. However for the purposes of this paper, and to simplify the translation of *ACSP* processes to existing verification tools such as FDR, we restrict the syntax of the language to *well-formed processes*.

Definition 4.4 (Well-Formed Processes). An *ACSP* process P is well-formed when:

Unique Names: Every location name in P is unique; i.e., every sub-term of P of the form $Q_1 \underset{E}{\parallel} Q_2$ has the property that $in(Q_1) \cap in(Q_2) = \emptyset$.

Flat Structure: Locations are not nested; i.e., every sub-term of P of the form $\Box_{i\in I}\, Q_i$, (if $e \le e'$ then Q else Q'), $(e{\rightarrow}Q)$, $(l!Q.Q')$, $(\mathrm{rec}X(\vec{y} := \vec{e}).Q)$ does not contain locations in Q, Q', Q_i.

Static Adaptation: Adaptation processes cannot be sent out processes containing higher-order events; i.e., every sub-term of P of the form $l!Q.R$ does not contain location outputs (adaptations actions) in Q.

Single Adaptation Procedure: Every location has at most one adaptation procedure; i.e., every sub-term of P of the form $Q_1 \underset{E}{\parallel} Q_2$ has the property that $out(Q_1) \cap out(Q_2) = \emptyset$. □

Well-formedness is preserved by the transition semantics, therefore, starting from well-formed processes we only reach well-formed processes.

Theorem 4.5. *If P is well-formed and $P \xrightarrow{e} P'$ then P' is well-formed.*

In the remaining sections we implicitly assume all processes are well-formed. As we show in the following section, these can model complex self-adaptive CPSs, such as the art gallery building. They can also be translated to FDR for verifying properties of such models (Section 6).

5 MODELLING THE ART GALLERY EXAMPLE

Here we discuss how the art gallery building can be encoded in *ACSP*. Following the steps described in Section 3, we first identify the cyber and physical components of the system and their topological relationships. For the art gallery, these were shown in Fig. 2.

The second step in our modelling technique involves examining each of the requirements of the CPS. Here we focus on Req. 1 from Section 2: *Visitors should not interfere with the restoration process.*

To enforce this requirement it would be sufficient to entirely disallow visitor access to the restoration area. However, as discussed, this is not desirable because of promotional reasons. Therefore, we will require instead that when visitors are in the restoration area a guard is present. We assume that the presence of a guard in *Corridor 2* is sufficient for ensuring rules of conduct are respected (e.g., visitors do not obstruct the work of restoration workers). Alternative approaches are also possible.

There are a number of ways that the system can implement the required presence of a guard. Using adaptation procedures for these implementations allows us to easily replace one with another in the model and compare them, while reusing the encoding of CPS components.

Here we examine two possibilities. In the first we encode a simple access policy: *the door between Corridor 2 and restoration area precludes entrance of visitors to the latter, unless a guard is in the corridor.*

The components relevant to this policy are the *Restoration Area* and *Corridor 2*. These components are connected in the model of Fig. 2 through the physical connectivity of a door. We design a family of first-order events to encode this connectivity: $t_{(c_2,\mathrm{ra})}$ and $t_{(\mathrm{ra},c_2)}$ respectively encode the movement of agents from *Corridor 2* (c_2) to the *Restoration Area* (ra) and *vice-versa*. Here t can take the form vis or grd to represent visitor or guard movement, respectively; for simplicity we only consider value vis.

The *Restoration Area* (including its door) can have one of two functionalities:

(1) Visitors are allowed to enter the restoration area, encoded by the process:

$$R_0 = vis_{(\mathrm{ra},c_2)}{\rightarrow}R_0 \;\Box\; vis_{(c_2,\mathrm{ra})}{\rightarrow}R_0$$

We assume that this process models an unlocked door with free movement of agents.

(2) Visitors are not allowed to enter the restoration area, encoded by the process: $R_1 = SKIP$. In this situation the door is assumed to be locked and agents need to present their credentials to open it.

The internal state of the *Restoration Area* keeps track of visitors movement to make sure that the number of people leaving are less than or equal to the number of people entering it. [1]

$$R(v) = (v > 0 \,\&\, vis_{(\mathrm{ra},c_2)}{\rightarrow}R(v-1)) \;\Box\; (vis_{(c_2,\mathrm{ra})}{\rightarrow}R(v+1))$$

We now define the adaptation procedure for the *Restoration Area*, which monitors movement of a guard from the *Stairs* to *Corridor 2*, and *vice-versa*, to allow/disallow access of visitors to the restoration area; this is:

$$\Pi_r = grd_{(s,c_2)}{\rightarrow}RA!R_0.\,\Pi_r \;\Box\; grd_{(c_2,s)}{\rightarrow}RA!R_1.\,\Pi_r$$

[1] we use $b\&P$ as a shorthand for if b then P else $SKIP$

In order for this adaptation to take effect, the process responsible for access to the restoration area should be in a location with name *RA*. The encoding of the restoration area, including its adaptation procedure, in the initial state where no guard or visitor is present in the upper floor thus becomes:

$$ResArea = (\nu\, RA)(\Pi_r \underset{E_v}{\|} R(0) \underset{E_v}{\|} RA\langle R_1\rangle)$$

where $E_v = \{vis_{(c_2,\mathrm{ra})}, vis_{(\mathrm{ra},c_2)}\}$.

The *Corridor 2* component allows movement to/from the *Restoration Area*, but also the *Stairs*, encoded by the family of events $t_{(c_2,s)}$ and $t_{(s,c_2)}$, where $t \in \{grd, vis\}$. The internal state of *Corridor 2* keeps track of the number of visitors in the space, which can be expressed as a monitor of the movement events.

$$C(v_{c_2}, v_{\mathrm{ra}}, g_{c_2}) = \Box \begin{cases} & vis_{(s,c_2)} \rightarrow C(v_{c_2}+1, v_{\mathrm{ra}}, g_{c_2}) \\ v_{c_2} > 0\ \& & vis_{(c_2,s)} \rightarrow C(v_{c_2}-1, v_{\mathrm{ra}}, g_{c_2}) \\ v_{c_2} > 0\ \& & vis_{(c_2,\mathrm{ra})} \rightarrow C(v_{c_2}-1, v_{\mathrm{ra}}+1, g_{c_2}) \\ v_{\mathrm{ra}} > 0\ \& & vis_{(\mathrm{ra},c_2)} \rightarrow C(v_{c_2}+1, v_{\mathrm{ra}}-1, g_{c_2}) \\ g_{c_2} = 0\ \& & grd_{(s,c_2)} \rightarrow C(v_{c_2}, v_{\mathrm{ra}}, g_{c_2}+1) \\ g_{c_2} > 0\ \& & grd_{(c_2,s)} \rightarrow C(v_{c_2}, v_{\mathrm{ra}}, g_{c_2}-1) \\ & cnt.v_{c_2}+v_{\mathrm{ra}} \rightarrow C(v_{c_2}, v_{\mathrm{ra}}, g_{c_2}) \end{cases}$$

Here the event *cnt* is used to query the value of the monitor from the main adaptation process.

For our policy to be correct, the guard should be allowed to leave the upper floor (through the stairs) only when there are no visitors left in it. Thus *Corridor 2* can have one of two functionalities:

(1) The guard is allowed to leave *Corridor 2* as there are no visitors in the upper floor, encoded as the process:

$$C2_0 = \Box_{t\in\{vis,grd\}} \begin{cases} t_{(s,c_2)} \rightarrow C2_0 \\ t_{(c_2,s)} \rightarrow C2_0 \end{cases} \Box\ vis_{(c_2,\mathrm{ra})} \rightarrow C2_0 \Box\ vis_{(\mathrm{ra},c_2)} \rightarrow C2_0$$

(2) The guard is not allowed to leave, encoded by the process:

$$C2_1 = \Box \begin{cases} vis_{(s,c_2)} \rightarrow C2_1 \\ vis_{(c_2,s)} \rightarrow C2_1 \\ vis_{(c_2,\mathrm{ra})} \rightarrow C2_1 \\ vis_{(\mathrm{ra},c_2)} \rightarrow C2_1 \\ grd_{(s,c_2)} \rightarrow C2_1 \end{cases}$$

The adaptation procedure of *Corridor 2* monitors the movements of visitors in and out of the *Stairs* and updates the door allowing/disallowing the guard to leave the floor.

$$\begin{aligned} \Pi_{c_2} = {} & vis_{(s,c_2)} \rightarrow cnt.v \rightarrow \text{if } v = 1 \text{ then } C2!C2_1.\,\Pi_{c_2} \text{ else } \Pi_{c_2} \\ & \Box\ vis_{(c_2,s)} \rightarrow cnt.v \rightarrow \text{if } v = 0 \text{ then } C2!C2_0.\,\Pi_{c_2} \text{ else } \Pi_{c_2} \\ & \Box\ vis_{(c_2,\mathrm{ra})} \rightarrow \Pi_{c_2} \\ & \Box\ vis_{(\mathrm{ra},c_2)} \rightarrow \Pi_{c_2} \\ & \Box\ grd_{(c_2,s)} \rightarrow \Pi_{c_2} \\ & \Box\ grd_{(s,c_2)} \rightarrow \Pi_{c_2} \end{aligned}$$

We can now write the encoding of *Corridor 2* which uses location *C2* to adapt the functionality of the door connecting it to the *Stairs*.

$$Corr2 = (\nu\, C2)(\nu\, cnt v)(\Pi_{c_2} \underset{E,cnt}{\|} (C(0) \underset{E}{\|} C2\langle C2_0\rangle))$$

Here E is the set of all movement events involving c_2.

In the composition of *Corr2* and *ResArea*, the two components interact explicitly through the door between them but the *Restoration Area* also listens implicitly to the guard movement in *Corridor 2*. These events have to synchronise across the components

$$Corr2 \underset{E_1}{\|} ResArea \qquad (\star)$$

where $E_1 = \{vis_{(\mathrm{ra},c_2)}, vis_{(c_2,\mathrm{ra})}, grd_{(s,c_2)}, grd_{(c_2,s)}\}$.

As we explain in the following section, we can verify the correctness of $(\star)$ *independently* of the rest of the system, through our translation to FDR.

We can also encode an alternative policy in which the door to the restoration area is rarely locked, improving the movement of restoration workers. To do this, we extend the adaptation procedure of the *Restoration Area*, so that it monitors the entry of visitors to the *Stairs* and calls the guard from *Corridor 1* when necessary.

$$\begin{aligned} \Pi_r(b) = {} & grd_{(s,c_2)} \rightarrow RA!R_0.\,\Pi_r(\mathtt{tt}) \\ & \Box\ grd_{(c_2,s)} \rightarrow RA!R_1.\,\Pi_r(\mathtt{ff}) \\ & \Box\ vis_{(c,s)} \rightarrow \text{if } \neg b \text{ then } call\_guard \rightarrow \Pi_r(b) \text{ else } \Pi_r(b) \\ & \Box\ vis_{(\mathrm{ra},c_2)} \rightarrow \Pi_r(b) \Box\ vis_{(c_2,\mathrm{ra})} \rightarrow \Pi_r(b) \end{aligned}$$

Note that the rest of the encoding of the system needs no change.

6 VERIFICATION PROCESS

As we discussed earlier, we can use existing verification technology to verify sets of components of self-adaptive CPSs modelled in our language, provided that they entirely encapsulate their adaptation procedures. However, our technique allows us to identify such sets of components in complex self-adaptive CPSs, and indeed the components in $(\star)$ do encapsulate their adaptation procedures.

Here we show how to use FDR for such verification. To do this we employ a translation from *ACSP* to CSP, which is the core language of FDR. First we briefly review CSP and FDR and then present the translation.

6.1 CSP and FDR

Refinement-checking is a verification technique whereby an implementation is deemed correct if its behaviour is contained in the specification's behaviour—it is a refinement of the specification. FDR [20] is a refinement-checker based on CSP. The CSP language consists of processes derived by the syntax:

$$\begin{aligned} P, Q ::= {} & e \rightarrow P \mid P \Box Q \mid P \underset{A}{\|} Q \mid SKIP \mid \text{if } b \text{ then } P \text{ else } P \\ & \mid \text{let } x = P \text{ within } Q \mid P[[e_2/e_1]] \mid P \triangle Q \mid P \setminus A \end{aligned}$$

Here $e \rightarrow P$ means that the process P is guarded by action e. CSP allows us to pattern-match e. External (deterministic) choice $P \Box Q$ allows the environment to choose between synchronising with P or Q. The processes if b then P else P, $P \setminus A$, $P[[e_2/e_1]]$, let $x = P$ within Q and *SKIP* represent conditional, action hiding, event renaming from e_1 to e_2, let declaration and the deadlocked

$$\textsc{tChx}\ \frac{i \in \mathcal{I} \text{ implies } \Gamma \vdash P_i \rhd S_i}{\Gamma \vdash \Box_{i\in\mathcal{I}}\, e_i{\rightarrow}P_i \rhd \Box_{i\in\mathcal{I}}\, e_i{\rightarrow}S_i} \qquad \textsc{tScp}\ \frac{\Gamma \vdash M \rhd S}{\Gamma \vdash (\nu e)M \rhd S \setminus \{e\}} \qquad \textsc{tRec}\ \frac{\Gamma \vdash P \rhd S}{\Gamma \vdash \mathrm{rec}X(\vec{y} := \vec{e}).P \rhd \mathrm{let}\, X(\vec{y}) = S \,\mathrm{within}\, X(\vec{e})}$$

$$\textsc{tIf}\ \frac{\Gamma \vdash P \rhd S \quad \Gamma \vdash Q \rhd T}{\Gamma \vdash \text{if } e_1 \le e_2 \text{ then } P \text{ else } Q \rhd \text{if } e_1 \le e_2 \text{ then } S \text{ else } T} \qquad \textsc{tSkp}\ \Gamma \vdash SKIP \rhd SKIP \qquad \textsc{tApp}\ \Gamma \vdash X(\vec{e}) \rhd X(\vec{e}) \qquad \textsc{tSnd}\ \frac{l \in \Gamma \quad m(l!P) = e \quad \Gamma \vdash Q \rhd S}{\Gamma \vdash l!P.Q \rhd e{\rightarrow}S}$$

$$\textsc{tLoc}\ \frac{l \in \Gamma \quad \Gamma \vdash P \rhd S}{\Gamma \vdash l\langle P\rangle \rhd S \vartriangle rec_\Gamma(l)} \qquad \textsc{tVar}\ \Gamma \vdash y \rhd y$$

Figure 4: Translation into CSP

processes, respectively. As in our language, interleaving $P \underset{A}{\|} Q$ requires P and Q to synchronise on actions in the set A but requires no synchronisation on events not in A. An interrupt process, written $P \vartriangle Q$, propagates any event from P without affecting the interrupt, but if Q ever performs a visible event then this removes the interrupt and P, and the entire process behaves as Q.

The three main semantic models of CSP are trace (T), stable failure (F) and failure-divergence (FD) models. In the trace model, $P \sqsubseteq_{\mathrm{T}} Q$ denotes that $traces(Q) \subseteq traces(P)$. This is useful for specifying safety properties, i.e., all the traces of implementation Q are in the traces of the specification P. The stable failure model compares the set of failures: $failures(Q) \subseteq failures(P)$. A failure is the possibility to refuse a set of actions after performing a trace. In this model, we can define liveliness properties, by showing that if an action is not refused (i.e., it eventually happens) in the specification P, then it is also not refused by implementation Q. Finally, the failure-divergence model further compares the set of diverging processes: $failures(Q) \cup divergence(Q) \subseteq failures(P) \cup divergence(P)$. Here a divergence is a trace that can lead to an infinite loop, thus refusing to perform any observable transition after the loop.

FDR [20] is an automatic refinement tool for machine readable CSP—CSP$_M$[33]. We can check that an implementation refines the specification according to one of the semantic models through assertions `assert Spec [M= Impl` where `M` ∈ {`T, F, FD`} We can also verify that the implementation is deterministic, deadlock free and livelock free (divergence free) according to one of the semantic models `assert Impl :[deadlock free [M]]`.

6.2 Translation into CSP

In Fig. 4, we depict the translation of our modelling language *ACSP* into CSP. This is defined by structural induction on *ACSP* terms, and presented by judgements of the form $\Gamma \vdash P \rhd S$ translating an *ACSP* process P to a CSP process S, with respect to an environment Γ which is a set of location names used in P.

By a pre-processing step, we can collect all location names used, and all processes inside higher-order outputs. This is possible because we work with well-formed processes (Definition 4.4). We can thus assume an injective map m mapping from higher-order prefixes to distinguished CSP events. We trivially extend this mapping first-order events, such that $m(e) = e$ for all events e. We also let p be the inverse mapping, taking events back to the process communicated i.e., $p(e) = P$ if there exists a location l where $m(l!P) = e$, or e otherwise. Furthermore, a function ch returns the set of events attached to each location l i.e., $ch(l) = \{e \mid \forall P.\, m(l!P) = e\}$.

The rules TChx, tScp, tRec, tIf, tSkp, tApp give a direct mapping to CSP of many *ACSP* processes. The adaptation mechanism is encoded in the rules tSnd, tLoc and tPar. Rule tSnd translates $l!P.Q$ by prefixing the translation of Q with the event defined in $m(l!P)$. Rule tLoc translates a location l, which is the receiving side of adaptation of l. We utilise the interrupt construct to implement the location: the translation of process P can be interrupted by any event in $ch(l)$. Here we use

$$rec_\Gamma(l) = \Box_{e\in ch(l)}\, e{\rightarrow}(T_e \vartriangle rec_\Gamma(l))$$

where any $e \in ch(l)$ translates to T_e by $\Gamma \vdash p(e) \rhd T_e$. This CSP interrupt unfolds $rec_\Gamma(l)$ with every $ch(l)$ event, guaranteeing the execution of the right (translated) process that should run after each adaptation of l, and reestablishing the interrupt.

Example 6.1 (Adaptation Processes). Assume the map m such that $m(l!a{\rightarrow}SKIP) = e_1$ and $m(l!b{\rightarrow}SKIP) = e_2$. Then we have the translation

$$\Gamma \vdash\ l\langle a{\rightarrow}b{\rightarrow}SKIP\rangle\ \rhd\ (a{\rightarrow}b{\rightarrow}SKIP) \vartriangle R$$

where $R = e_1{\rightarrow}(a{\rightarrow}SKIP \vartriangle R) \,\Box\, e_2{\rightarrow}(b{\rightarrow}SKIP \vartriangle R)$ for all Γ such that $l \in \Gamma$.

Dually, the process $l!a{\rightarrow}SKIP.SKIP$ initiating an adaptation translates to CCS according to: $\Gamma \vdash l!a{\rightarrow}SKIP.SKIP\ \rhd\ e_1{\rightarrow}SKIP$. □

Finally, rule tPar translates a parallel composition $M \underset{E}{\|} N$ into a CSP parallel composition. The set of events E is transferred to the CSP parallel, extended with synchronisations of events encoding adaptation between M and N. This is expressed by the set $(in(N) \cap out(M)) \cup (in(M) \cap out(N))$ in the premises of the rule. The translation of M and N is then done under an environment containing these extra names, which can be used for translating locations and higher-order prefixes in them. The events corresponding to these location names are then localised around the parallel. This prevents the interruption of (translated) locations if there are no corresponding prefix processes.

Example 6.2 (Adaptation). The processes in Example 6.1 can be composed together using tPar to model a complete adaptable system. Note that the chosen L required by the rule need to contain

at least l as $l \in out\,(l!a{\rightarrow}SKIP.SKIP) \cap in\,(l\langle a{\rightarrow}b{\rightarrow}SKIP\rangle)$. It follows

$$\Gamma' \vdash \left(l\langle a{\rightarrow}b{\rightarrow}SKIP\rangle \underset{\emptyset}{\parallel} l!a{\rightarrow}SKIP.SKIP\right) \rhd \left((a{\rightarrow}b{\rightarrow}SKIP) \triangle R \underset{\{e_1,e_2\}}{\parallel} e_1{\rightarrow}SKIP\right) \setminus \{e_1, e_2\}$$

Note that Γ' must not contain l as this is added in the rule tPar, which essentially guarantees that there is no other process that initiate adaptation on l, and $\{e_1, e_2\}$ are the events identifying potential processes communicated on l. □

We prove that the translation, depicted in Fig. 4, is a strong bisimulation. That is, transitions of the *ACSP* term are in loc-step with the corresponding transitions of the CSP translation.

Theorem 6.3. *Let $\Gamma \vdash M \rhd S$; then:*

(1) If $M \xrightarrow{\alpha} M'$ then there exists S' such that $S \xrightarrow{m(\alpha)} S'$ and $\Gamma \vdash M' \rhd S'$
(2) If $S \xrightarrow{e} S'$ then there exist M' and α such that $m(\alpha) = e$ and $M \xrightarrow{\alpha} M'$ and $\Gamma \vdash M' \rhd S'$

This theorem allows us to conclude that every property of the *ACSP* processes is also a property of the translated CSP processes, and vice-versa. Thus reasoning in FDR about the translated processes leads to verification results about the original *ACSP* processes, and therefore verification results of the system.

6.3 Evaluation of the verification technique

Translating the system components of the *Restoration Area* and *Corridor 2*, modelled in our language as process (★) in Section 5, leads to an FDR file of 115 lines of code. Many of the transitions are internal transitions, which FDR can eliminate using a simplification command. This leads to a system of 27 states which can be easily verified by the tool on a personal computer with 8GB of memory. When translating the entire model of the art gallery to FDR we get a model that quickly runs out of memory on the same system, even with the use of simplification. It is thus clear that compositional verification is a valuable technique for providing formal assurances for self-adaptive CPSs.

The verification of (★) was done by encoding simple specification automata in CSP accepting a language of correct traces, and then showing that the translation of (★) has a subset of these traces using FDR's trace assertions. One of these specification processes does not include traces where a visitor accesses the restoration area without a guard present. Another shows that the guard does not leave the *Corridor 2* if there are still visitors in the second floor.

7 RELATED WORK

In this section we discuss existing approaches to model and verify self-adaptive (SA) systems.

Refinement-based models. Hachicha et al. [23] present a refinement-based framework for modelling and verifying SA systems. A SA system is specified using UML diagrams, which are translated automatically into Event-B. The resulting model can be verified using the Rodin theorem prover. Göthel et al. [21] overview how to model different design patterns for SA systems using CSP. Bartels et al. [3] study how SA systems can be modelled using CSP. The framework explicitly separates adaptive and non-adaptive behaviour, modelled through events and processes. The authors discuss the limitation of using CSP for modelling dynamic adaptation which stems from the inability to create processes or events on-the-fly. The dynamic creation of processes and events can be simulated if the events can be determined at design time and utilise guards to switch between events. This work was extended by Göthel et al. [22] to include time dependencies within the model. In our work, indeed, adaptation processes can be determined at translation time and we are able to use FDR for verification. Our language allows dynamic generation of processes but more advanced verification tools would be necessary to reason about such behaviour. Our work focuses on compositional verification, which the above works do not address. Moreover, our framework enables the exploration of alternative adaptation procedures, which can be a valuable tool for designing and verifying large-scale SA CPSs.

Hierarchical models. Bruni et al. [9] use Maude to model each step in the MAPE-K feedback loop as an abstract state machine (ASM), which can be model-checked using PVesta. Iftikhar et al. [26] use timed automata and timed computational tree logic (TCTL) to model decentralized SA systems and specify temporal safety and liveness properties. These properties are model-checked using Uppaal. Klarl et al. [28] present hierarchical LTS (H-LTS) which can be model checked with SPIN and translated automatically into Java code. Zhao et al. [38] model SA systems using mode-automata and define mode extended LTL (*m*LTL); an extension of LTL with context-dependent formulas for the specification of the system. The semantics of *m*LTL is derived from that of LTL. Zhao et al. used the NuSMV model checker as a verification tool. Jalili et al. [1] explore a method to model and verify at runtime decentralized SA systems based on HPobSAM framework, taking advantage of decentralisation to achieve compositionality. In these works, the ability to decompose systems into independently verifiable components is limited or non-existent. Our work provides a structured method to do this, when requirements allow it, even in systems with components that are intricately linked with cyber and physical relationships.

Petri-Nets. Zhang et al. [37] model SA systems as a collection of petri-nets. Each petri-net has a single initial and final state. A SA system transition between petri-nets through these states. This work has been extended in [10] to incorporate temporal constraints by considering the time-based petri-nets [5]. In a similar fashion, Context Petri-Nets (CPN), introduced by Cardozo et al. [11], allow dynamic reconfiguration of petri-nets to model adaptation; CPN can be then translated automatically into petri-nets. Ding et al. [17] explain how neural networks can be utilised to implement the adaptation function wiring petri-nets together. Petri-nets are not easily decomposable, and thus compositional verification is hard to achieve. Moreover, they do not provide a fertile ground to explore alternative adaptation procedures at different granularities of the system, as precise adaptation procedures are hard to encode and modify independently of the base system model.

Process languages with passivation. Passivation has been introduced in process calculi to enable the encoding of complex distributed systems with components that can be stopped and resumed [34]. Bavetti et al. [8] propose the $\mathcal{E}$-calculus, a higher-order calculus inspired by the Calculus of Communicating Systems (CCS) without restriction and renaming. A process P which is adaptable and located at a, is denoted as $a[P]$. The environment can at any time communicate a context to a, installing the context at the a-location. This context may have holes which are then filled by copies of P. This results in a highly dynamic language; verification in this language would require new special-purpose techniques and tools to be developed. Moreover, the absence of location restriction (scoping) severely limits compositional reasoning in this language. Our framework is based in a more tamed higher-order language, which enables a translation to existing verification tools based on first-order languages, as we have shown with our translation to FDR. We make use of location restriction which allows us to consider parts of the system where all locations are locally scoped. Such parts can be verified independently from the rest of the system, thus enabling useful compositional verification. We also utilise CSP-style multi-party synchronisation rather than CCS binary communication, to improve the separation of adaptation procedures, which must monitor the events in the system, from the system itself.

Other process languages. Debois et al. [16] define the DCR process language, a Turing-complete declarative process language to model and verify runtime adaptation in a modular fashion. They define a non-invasive adaptation in a decidable fragment of the language. An adaptation is non-invasive if a new process is spawned during the adaptation. This sub-language is able to represent systems where a new resource or new condition is introduced during adaptations. Lochau et al. [29] define DeltaCCS, an extension of CCS to explicitly model behavioural variability in processes. The authors also implemented a DeltaCCS model checker to verify the processes. In our work, the process language is based on CSP-style synchronisation to achieve a better separation of concerns as base system components are oblivious to the adaptation procedures that may affect them. Moreover, our process language, based on higher-order communication can naturally encode a larger class of systems. Bono et al. [6] present a data-driven approach to modelling SA systems based on session types. Adaptation does not alter the process but filters the communication between the component from the rest of the system. Our approach is different as it is based on a more natural encoding of self-adaptation with higher-order communications, giving us more flexibility to encode systems. We also use scoped locations, instead of session types to achieve compositional reasoning, which enables us to leverage existing verification tools such as CSP. Schroeder et al. [35] model SA systems compositionally using an assume-guarantee framework. This requires special-purpose verification technology which is still to be developed [13, 27]. Our framework leverages existing verification technology, such as FDR. However, it would be interesting to explore possibilities of combining the two frameworks.

Other verification tecniques for self-adaptive CPSs. Tsigkanos et al. [36] use bigraphical reactive systems to represent topological relationships of cyber-physical systems. They apply explicit state model checking to reason about security threats brought by changes in the topological relationships that may occur at runtime. They also provide an automated planning technique to identify security controls to prevent or mitigate discovered threats. However, adoption of model checking to perform security analysis may lead to state explosion and may not be computationally feasible for large-scale systems. To address this problem, Filieri and Tamburrelli [19] describe a number of strategies (i.e. state elimination [14] and algebraic approaches [18]) for using probabilistic model checking at runtime when the system behaviour is expressed as a Discrete Time Markov Chain (DTMC). These strategies can only be applied when changes in the system representation can be expressed as different assignments to its parameters (e.g., different probabilities for the transitions of the DTMC) and cannot cope with changes in the structure or behaviour of the system and its operating environment. Balasubramaniyan et al. [2] model SA CPSs using timed-automata and verify them in the UPPAAL model checker. The model incorporates explicit time delays that differ between physical and cyber components. None of these techniques provide a compositional method for the verification of SA systems.

8 CONCLUSIONS

This paper provides three main contributions. First, it proposes *Adaptive CSP*, a novel modelling language to represent self-adaptive CPSs in a modular way, separating system actions from adaptation actions. Then, it provides a technique to use topological relationships of CPS to decompose the system into a small set of components that can affect—independently from the rest of the system—satisfaction of a given requirement. A self-adaptation procedure aiming to satisfy a given requirement can be localised to the components that affect its satisfaction. As the state space of these components is typically smaller compared to that of the rest of the system, we can use formal verification technology, such as FDR, to check that the component behaviour and the self-adaptation procedure identified by the system designer can satisfy a given set of requirements. We also provide a translation from a subset of our language to FDR to perform verification of self-adaptive CPSs. We evaluated our approach using a substantive art gallery example. Our results demonstrate that our approach has the befit of reducing the memory and time required to verify properties of the self-adaptive CPS. Our technique for discovering a good level of granularity for an adaptation procedure that ensures satisfaction of system requirements can reduce the size of components that need to be verified.

In future work we are planning to extend our approach to improve its usability. We aim to develop a graphical interface to support the system designer in encoding of a system into *Adaptive CSP*. Moreover we want to integrate automated planning techniques to support the automatic generation of alternative adaptation strategies in our framework. We also hope to explore other verification technology that can deal with the advanced features of *Adaptive CSP*.

REFERENCES

[1] B. Abolhasanzadeh and S. Jalili. 2016. Towards modeling and runtime verification of self-organizing systems. *Expert Systems with Applications* 44, Supplement C (2016), 230 – 244.

[2] S. Balasubramaniyan, S. Srinivasan, F. Buonopane, B. Subathra, J. Vain, and S. Ramaswamy. 2016. Design and verification of Cyber-Physical Systems using TrueTime, evolutionary optimization and UPPAAL. *Microprocessors and Microsystems* 42 (2016), 37 – 48.

[3] B. Bartels and M. Kleine. 2011. A CSP-based framework for the specification, verification, and implementation of adaptive systems. In *Proc. 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 158–167.

[4] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. 1995. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III (Lecture Notes in Computer Science)*. Springer–Verlag, 232–243.

[5] B. Berthomieu and M. Diaz. 1991. Modeling and Verification of Time Dependent Systems Using Time Petri Nets. *IEEE Trans. Softw. Eng.* 17, 3 (March 1991), 259–273.

[6] V. Bono, M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. 2017. Data-driven adaptation for smart sessions. *Journal of Logical and Algebraic Methods in Programming* 90, Supplement C (2017), 31 – 49.

[7] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel. 2015. Morph: A Reference Architecture for Configuration and Behaviour Self-adaptation. In *Proc. 1st International Workshop on Control Theory for Software Engineering*. ACM, 9–16.

[8] M. Bravetti, C. Di Giusto, J. A. Pérez, and G. Zavattaro. 2012. Adaptable processes. *Logical Methods in Computer Science* 8, 4 (2012).

[9] R. Bruni, A. Corradini, F. Gadducci, A. Lluch Lafuente, and A. Vandin. 2015. Modelling and analyzing adaptive self-assembly strategies with Maude. *Science of Computer Programming* 99 (2015), 75–94.

[10] M. Camilli, A. Gargantini, and P. Scandurra. 2015. Specifying and verifying real-time self-adaptive systems. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 303–313.

[11] N. Cardozo, S. Gonzalez, K. Mens, R. Van Der Straeten, and T. DHondt. 2013. Modeling and Analyzing Self-Adaptive Systems with Context Petri Nets. In *2013 International Symposium on Theoretical Aspects of Software Engineering*. IEEE, 191–198.

[12] R. Cleaveland, J. Parrow, and B. Steffen. 1993. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Trans. Program. Lang. Syst.* 15, 1 (1993), 36–72.

[13] A. David, K. G. Larsen, A. Legay, M. H. Møller, U. Nyman, A. P. Ravn, A. Skou, and A. Wąsowski. 2012. Compositional verification of real-time systems using ECDAR. *International Journal on Software Tools for Technology Transfer* 14, 6 (2012), 703–720.

[14] C. Daws. 2004. Symbolic and Parametric Model Checking of Discrete-Time Markov Chains. In *Proc. 1st International Colloquium on Theoretical Aspects of Computing*, Vol. 3407. Springer, 280–294.

[15] R. De Nicola and M. Hennessy. 1984. Testing Equivalences for Processes. *Theor. Comput. Sci.* 34 (1984), 83–133.

[16] S. Debois, T. Hildebrandt, and T. Slaats. 2015. *Safety, Liveness and Run-Time Refinement for Modular Process-Aware Information Systems with Dynamic Sub Processes*. Springer International Publishing, 143–160.

[17] Z. Ding, Y. Zhou, and M. Zhou. 2016. Modeling Self-Adaptive Software Systems With Learning Petri Nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 46, 4 (apr 2016), 483–498.

[18] A. Filieri, C. Ghezzi, and G. Tamburrelli. 2011. Run-time Efficient Probabilistic Model Checking. In *Proc. 33rd International Conference on Software Engineering*. 341–350.

[19] A. Filieri and G. Tamburrelli. 2013. Probabilistic Verification at Runtime for Self-Adaptive Systems. *Assurances for Self-Adaptive Systems* 7740 (2013), 30–59.

[20] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A.W. Roscoe. 2014. FDR3 — A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, E. ĂĄbrahĂąm and K. Havelund (Eds.), Vol. 8413. 187–201.

[21] T. Göthel and B. Bartels. 2015. *Modular Design and Verification of Distributed Adaptive Real-Time Systems*. Springer International Publishing, 3–12.

[22] T. Göthel, N. Jähnig, and S. Seif. 2017. *Refinement-Based Modelling and Verification ofÂăDesign Patterns forÂăSelf-adaptive Systems*. Springer International Publishing, 157–173.

[23] M. Hachicha, R. B. Halima, and A. H. Kacem. 2016. Modeling and verifying self-adaptive systems: A refinement approach. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 003967–003972.

[24] M. Hennessy and R. Milner. 1985. Algebraic Laws for Nondeterminism and Concurrency. *J. ACM* 32, 1 (1985), 137–161.

[25] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677.

[26] M. U. Iftikhar and D. Weyns. 2012. A Case Study on Formal Verification of Self-Adaptive Behaviors in a Decentralized System. In *Proc. 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012 (EPTCS)*, N. Kokash and A. Ravara (Eds.), Vol. 91. 45–62.

[27] P. Inverardi, P. Pelliccione, and M. Tivoli. 2009. Towards an assume-guarantee theory for adaptable systems. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on*. 106–115.

[28] A. Klarl. 2015. Engineering Self-Adaptive Systems with the Role-Based Architecture of Helena. In *2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE, 3–8.

[29] M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. 2014. *DeltaCCS: A Core Calculus for Behavioral Change*. Springer Berlin Heidelberg, 320–335.

[30] R. Milner. 1989. *Communication and concurrency*. Prentice Hall.

[31] L. Pasquale, C. Ghezzi, C. Menghi, C. Tsigkanos, and B. Nuseibeh. 2014. Topology Aware Adaptive Security. In *Proc. 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 43–48.

[32] D. Sangiorgi and D. Walker. 2001. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press.

[33] B. Scattergood. 1998. *The semantics and implementation of machine-readable CSP*. Ph.D. Dissertation. Citeseer.

[34] A. Schmitt and J.-B. Stefani. 2004. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing, IST/FET International Workshop, GC 2004, Rovereto, Italy, March 9-12, 2004, Revised Selected Papers (Lecture Notes in Computer Science)*, C. Priami and P. Quaglia (Eds.), Vol. 3267. Springer, 146–178.

[35] A. Schroeder, S. S. Bauer, and M. Wirsing. 2011. A contract-based approach to adaptivity. *Journal of Logic and Algebraic Programming* 80, 3-5 (apr 2011), 180–193.

[36] C. Tsigkanos, L. Pasquale, C. Ghezzi, and B. Nuseibeh. 2017. On the Interplay Between Cyber and Physical Spaces for Adaptive Security. *IEEE Transactions on Dependable and Secure Computing* PP, 99 (2017).

[37] J. Zhang and B. H. C. Cheng. 2006. Model-based development of dynamically adaptive software. In *Proceeding of the 28th international conference on Software engineering, ICSE '06*. ACM Press, 371.

[38] Y. Zhao, D. Ma, J. Li, and Z. Li. 2011. Model Checking of Adaptive Programs with Mode-extended Linear Temporal Logic. In *2011 Eighth IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*. IEEE, 40–48.